Object-Oriented Modeling and Design of Database Federations

H. Balsters

University of Groningen
Faculty of Management and Organization
P.O. Box 800
9700 AV Groningen, The Netherlands

h.balsters@bdk.rug.nl

SOM theme A Primary processes within firms

Abstract

We describe a logical architecture and a general semantic framework for precise specification of so-called database federations. A database federation provides for tight coupling of a collection of heterogeneous component databases into a global integrated system. Our approach to database federation integrates in a uniform and systematic manner the underlying database schemas of the component legacy systems to a separate, newly defined integrated database schema. This integrated database is completely virtual, and will constitute the actual federated database. That is, queries posed against the federated system will be posed against this virtual integrated database; these global queries will then be mapped by the mediator to actual local queries against the existing (legacy) component databases. Our approach is based upon the UML/OCL data model. UML is the *de facto* standard language for analysis and design in object-oriented frameworks, and is being employed more and more for analysis and design of Information systems, in particular information systems based on databases and their applications. Database specifications often involve specifications of constraints, and the Object Constraint Language (OCL) - as part of UML - can aid in the unambiguous modelling of database constraints. One of the central notions in database modelling and in constraint specifications is the notion of a database view; a database view closely corresponds to the notion of derived class in UML. We will employ OCL and the notion of derived class as a means to treat (inter-)database constraints and database views in a federated context. We will also offer a transaction model for a simple set of updates in database federations. The paper will demonstrate that our particular mediating system integrates component schemas without loss of constraint information. Furthermore, we will discuss a mapping of database specifications in terms of UML/OCL to the relational model.

1. Introduction

Modern information systems are often distributed in nature. Data and services are often spread over different component systems wishing to cooperate in an integrated setting. Cooperation of component systems in one integrated information system is becoming more and more important since information is often spread over different databases in one organization (or even spread over different organizations). Such information systems involving integration of cooperating component systems are called *federated information systems*; if the component systems are all databases then we speak of a federated database system (FDB). In current applications, there is more and more a tendency *not* to develop stand-alone, monolithic database systems; rather, the tendency is to employ existing (legacy) components by letting them work together in a single integrated environment. This tendency to build integrated, cooperating systems is often encountered in applications found in EAI (Enterprise Application Integration), which typically involve several, usually autonomous, component (data and service repositories) systems, with the desire to query and update information on a global, integrated level. In this paper we will address the situation where the component systems are so-called legacy systems; i.e. systems that are given beforehand and which are to interoperate in an integrated single framework in which the legacy systems are to maintain as much as possible their respective autonomy.

A major obstacle in designing interoperability of legacy systems is the heterogeneous nature of the legacy components involved. This heterogeneity is caused by the design autonomy of their owners in developing such systems. Legacy systems were typically designed to support local requirements, under constraints imposed by local rules, and often without taking into account any future cooperation with other systems. To address the problem of interoperability the term *mediation* has been defined [Wie95]. A database federation can be seen as a special kind of mediation, where all of the data sources are (legacy) databases, and the mediator offers a mapping to a (virtual)

DBMS-like interface. This interface offers the application the possibility to approach the federation via this integrated virtual database, which offers the user the illusion that he is interacting with an actual homogeneous, monolithic database. The mediator then maps queries against this virtual integrated database on to actual component databases. In our paper we will consider a *tightly-coupled approach* to database mediation, in which a global integrated schema of the federation is maintained, which can be accessed by a global query language. We base our notion of querying on the "*Closed World Assumption*" (CWA, [Rei84]), where the integrated database is to hold -in some manner- the "union" of the data in the underlying component databases. Central theme in our approach is that the integrated database on the federated level is completely *virtual*. The user of the federated system is offered the illusion that he is working with a monolithic homogeneous database system, while in fact this system basically resembles an interface, mapping interactions on the federated level to actions on the existing local database components. More precisely, the federated database will consist of an integrated *database view* on top of the existing legacy database components. For an overview of work on the virtual approach to database federation, we refer to [Hull97].

We concentrate on problems concerning integration of component legacy schemas on the level of the mediator. Schema integration requires the definition of relationships between schema elements of component systems. Detection and definition of such relationships can be heavily complicated by so-called *semantic heterogeneity* [DKM93,GSC96, Ver97]. Semantic heterogeneity refers to disagreement about the meaning, interpretation, or intended use of related data. It has been widely agreed upon that schema integration cannot be fully automated [ShL90], as this would require full knowledge of the semantics of the component schema elements. In order to tackle the problem of integrating semantic heterogeneity, we employ the UML/OCL data model. UML/OCL offers a high-level specification language and is

equipped with a unique combination of high expressiveness with a large degree of precision. UML is the *de facto* standard language for analysis and design in object-oriented frameworks, and is being employed more and more for analysis and design of Information systems, in particular information systems based on databases and their applications. Database specifications often involve specifications of constraints, and the Object Constraint Language (OCL) - as part of UML - can aid in the unambiguous modelling of database constraints. One of the central notions in database modelling and in constraint specifications is the notion of a *database view*, where a database view closely corresponds to the notion of derived class in UML. We will employ OCL and the notion of derived class as a means to treat database constraints and database views in a federated context. In [Bal02] it is demonstrated that the notion of derived class can be given a formal basis in OCL, and that derived classes in OCL have the expressive power of the relational algebra. Hence, OCL has the explicit power to emulate basic features of the relational query language SQL. The paper will demonstrate that our particular mediating system integrates component schemas without loss of constraint information; i.e., no loss of constraint information available at the component level may take place as result of integrating on the level of the virtual federated database. We will treat integration conflicts in a tightly-coupled environment, and show how to solve them by introducing a so-called *integration isomorphism*. This isomorphism will support the Closed World Assumption for database federations by correctly mapping a collection legacy databases to a virtual integrated database. Key to establishing this integration isomorphism, is the construction of a so-called *homogenizing function*; the homogenizing function (cf. [BB01]) maps schemas of component databases to the schema of the integrated database.

The only assumption that we make in this paper is that all legacy component databases have schemas that –somehow- are able to be (re-)modelled in terms of the UML/OCL language. This is a modest assumption, since most commercially available database systems (hierarchical, network, or relational) have schemas that are easily expressible in terms of the UML/OCL data model.

Our paper demonstrates how to specify and evaluate queries on the global level of the virtual integrated database, and how these queries decompose into local queries on the component databases. We also consider database updates in a federated context, and offer the basics of a transaction model for a simple set of updates in tightly-coupled database federations.

The paper includes a section on implementation issues. Following the approach offered in [Bal02] we have in principle a mapping of queries posed against a federated database (specified in terms of derived classes in UML/OCL) to SQL-code, thus providing the link to actual database implementations. Our paper also contains a discussion on federated database architectures, in which we demonstrate that *federated* database architectures can very much stay in line with the traditional three-level architecture for monolithic databases. Our paper ends with a discussion on methodology and heuristics for federated database design.

2. UML/OCL as a specification language for databases

Information systems, and in particular information systems based on databases and their applications, rely heavily on sound principles of analysis and design. This paper focuses on particular principles of analysis and design related to database applications. Following [BP98], we can state that object-oriented (OO) modelling can prove to be very beneficiary in (relational) database applications. A database is a permanent, self-descriptive repository of data stored in files. A database is self-descriptive in the sense that it not only contains the data, but also a description of the

data structure, or *schema*. In databases, the data usually change rapidly, while the schema stays relatively static. A database management system (DBMS) consists of software managing access to the data. DBMSs provide generic functionality for a broad range of applications; one of the foremost features of a DBMS is the availability of a *query language* offering an interactive means for reading and writing data from the database. A relational database has data represented as tables, and a relational DBMS manages access to tables of data and associated structures in a highly effective and efficient manner. (Relational databases use SQL as a data manipulation language, and tables are called *relations* in SQL.) Relational database applications can benefit substantially from OO modelling. The OO paradigm provides a uniform framework for both the design of database code and programming code. Database and their applications can thus be developed in one and the same conceptual framework. In fact, one can say that integrating relational databases into object-oriented applications is state of the art in software development practice. OO data models offer high-level modelling primitives leading to clear and concise specifications of database schemas. A high-level description of a database schema in terms of an OO data model can easily be mapped to a relational database schema employed by a conventional relational DBMS [BP98]. Hence, the analysis and design stage of a (relational) database can be separated in a clear and meaningful fashion.

The most important OO modeling language is UML, being the de facto standard for OO analysis and design of information systems [OMG99]. Recently, researchers have investigated possibilities of UML as a modeling language for (relational) databases. [BP98] describes in length how this process can take place, concentrating on schema specification techniques. [DH99, DHL01] investigate further possibilities by employing OCL (the Object Constraint Language [WK99]) for specifying constraints and business rules within the context of relational databases. The idea is that OCL provides expressiveness in terms of relatively abstract set definitions that should

prove to be sufficient to capture the general notion of (relational) database view. This idea of employing abstract object-oriented set definitions to captures views and constraints has also been pursued on the full level of object-oriented databases, be it not in the context of UML/OCL language, but rather in the context of an experimental OODB user language in combination with an underlying theoretical semantics [BBZ93, BV92]. In the more specific context of relational databases and OCL, [DH99] offer a framework for representing constraints within the relational data model. Some researchers take a very general approach investigating possibilities of UML/OCL; e.g., [AB01] treat OCL as a general query language for UML data models, and [EP00] use OCL as a general language for business modeling. Current research, however, has not yet shown an effective way to deal with an important aspect of (relational) database modeling, namely modeling of so-called database views. A (database) view is a derived table (or derived relation, in SQL), meaning that a view does not exist as a physical relation; rather a view is defined by an expression much like a query [GUW02]. Views, in turn, can be queried as if they existed physically, and in some cases, we can even modify view content. That is, a user is offered the impression that a view is some base relation inside the database, but in fact it is a derived (or virtual) relation defined in terms of the actual base relations constituting the database. View definitions are an important asset in database applications, because users are usually only interested in a part of the database, and not in the complete underlying corporate database. Hence, it is important that users have access to that part of the database considered relevant for their category of database applications. Our application area for views is focused on Federated Databases, where legacy databases are to interoperate by employing a so-called mediating system. This mediating system can be considered as an integration of a set of certain database views defined on the component legacy database systems.

Database views and query languages are strongly related, since views basically are no more than named queries. [GR97] is one of the first papers to investigate the possibilities of a general query language for UML; further investigations can be found in [AB01] and [MC99]. [AB01] have attempted to demonstrate that OCL can offer the basis for a general query language for UML data models by showing how to represent Cartesian products and projections in OCL, thus paving the way to the claim that OCL has the same expressive power as the so-called relational algebra [D00, GUW02]. By demonstrating such a result, one could also claim to have a basis for representing views within OCL. In [Bal02] it is demonstrated that the expressiveness of OCL actually includes that of the relational algebra. This is done by showing how to offer the notion of *derived class* a formal basis within the framework of UML/OCL, and subsequently using this notion of derived class to represent the notions of Cartesian product and (relational) join. This result establishes that OCL includes the expressiveness of the relational algebra, without resorting to language extensions of OCL. Once it is established that OCL includes the expressiveness of the relational algebra, then we also have provided a basis for representing the general notion of (relational) database view.

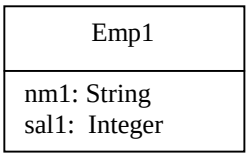
A derived class is a device for denoting a virtual class, defined in terms of already existing (base) classes (and possibly other derived classes). Views can be queried independently, with a semantics explained entirely in terms of queries on base classes. [Bal02] also offers a mapping to SQL-code [D00, GUW02], providing implementation support for our approach.

3. Basic principles: Databases and views in UML/OCL

Databases are basically a set of related tables. Tables in UML are represented by classes. Classes have attributes and corresponding domain values, while we can also have complex-valued attributes (i.e. non-first normal form) in UML by allowing for

enumerated sets as domains for attributes, and to employ UML-style relations to represent directly references to other objects in tables without residing to foreign-key constructs (to indirectly enforce this kind of modelling facility). Views, as derived tables, can also be represented in UML, which we will describe below.

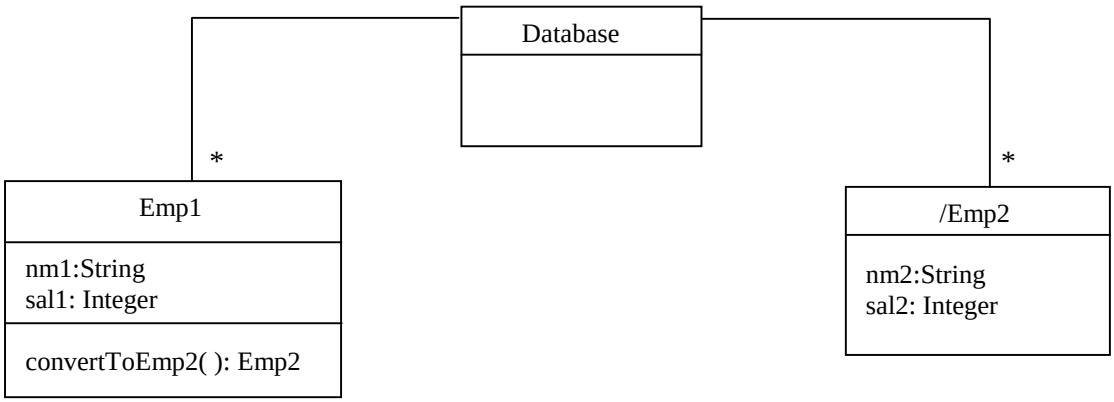
Let's consider the case that we have a class called Emp1 with attributes nm1 and sal1, indicating the name and salary of an employee object belonging to class Emp1



Now consider the case where we want to add a class, say Emp2, which is defined as a class whose objects are completely derivable from objects coming from class Emp1. The calculation is performed in the following manner. Assume that the attributes of Emp2 are nm2 and sal2 respectively (indicating name and salary attributes for Emp2 objects), and assume that for each object e1:Emp1 we can obtain an object e2:Emp2 by stipulating that e2.nm2=e1.nm1 and e2.sal2=(2 * e1.sal1). By definition the total set of instances of Emp2 is the set obtained from the total set of instances from Emp1 by applying the calculation rules as described above. Hence, class Emp2 is a *view* of class Emp1, in accordance with the concept of a view as known from the relational database literature. In UML terminology [BP98], we can say that Emp2 is a *derived class*, since it is completely derivable from other already existing class elements in the model description containing model type Emp1.

We will now show how to faithfully describe Emp2 as a derived class in UML/OCL in such a way that it satisfies the requirements of a (relational) view. First of all, we must satisfy the requirement that the set of instance of class Emp2 is the result of a calculation applied to the set of instances of class Emp1. The basic idea is that we

introduce a class called Database that has associations to classes Emp1 and Emp2. A database object will reflect the actual state of the database, and the system class Database will only consist out of one object in any of its states. Hence the variable *self* in the context of the class Database will always denote the actual state of the database that we are considering. In the context of this database class we can then define the calculation obtaining the set of instances of Emp2 by taking the set of instances of Emp1 as input.



Note that we have used a prefix-qualification by adding a slash to Emp2 indicating that Emp2 is a derived class definition [BP98]. Moreover, we have added an operation, called convertToEmp2, meant to coerce an arbitrary Emp1-object to an Emp2-object. This operation can be defined by the following OCL-specification

```
context   Emp1::convertToEmp2( ): Emp2
post:     self.convertToEmp2.nm2 = self.nm1  and
          self.convertToEmp2.sal2 = (2*self.sal1)
```

We now have all the ingredients necessary to specify the relation coupling the derived class Emp2 to the original class Emp1. This is done by including an invariant specification in the class Database telling us how to calculate the set of instances of Emp2 from the set of instances of Emp1

```
context Database inv:
self.Emp2 = self.Emp1→ collect(e:Emp1 | e.convertToEmp2) and
Emp1.allInstances = self.Emp1 and
Emp2.allInstances = self.Emp2
```

In this way we explicitly specify Emp2 as the result of a calculation performed on Emp1, and we also stipulate that the only Emp1- and Emp2-objects in the database are those obtained from the links starting from the database-object *self*.

Discussion: How ***not*** to represent views

A reader might have the idea that there is an alternative (and rather simple) way to define database views in UML/OCL employing constraints, and without having to introduce the notion of derived class. We wish to discuss this topic here, because it deals with somewhat widespread misconception of what a database view actually is. Consider our example of Emp2 as a database view derived from the base class Emp1. One might be inclined to think that Emp2 could also be defined indirectly by employing suitable constraints. For example, one could introduce Emp2 as an extra model type (hence not as a derived class), and then stipulate the following two constraints

```
context Emp2 inv:
Emp1.allInstances →
exists(e1 | e1.nm1 = self.nm2 and 2*e1.sal1 = self.sal2)
```

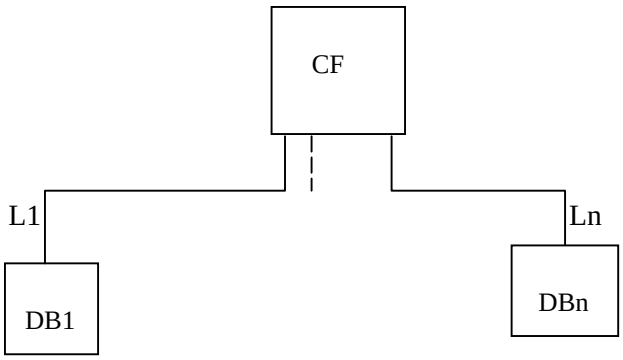
```
context  Emp1  inv:
Emp2.allInstances →
exists(e2 | e2.nm2 = self.nm1 and e2.sal2 = 2*self.sal1)
```

This way the content of class Emp2 -seemingly- is defined as the desired content of class Emp1, with appropriately changed values for the name and salary components. The thing that is wrong with this approach is that this does not constitute a view definition. This approach rather defines two autonomous base classes that are constrained by one another, and it does not reflect the desired result that Emp2 is a *virtual* class with content that is *derived* from class Emp1 *by calculation.* That is, the desired situation is the one where Emp1 can freely change its contents (due to updates performed by users of the database), *irrespective* of the content of Emp2; the content of the virtual class Emp2 should then be deducible on demand and at any given moment by performing a suitable calculation on the content of Emp1. This reflects the situation that a view is basically no more than a named query result.

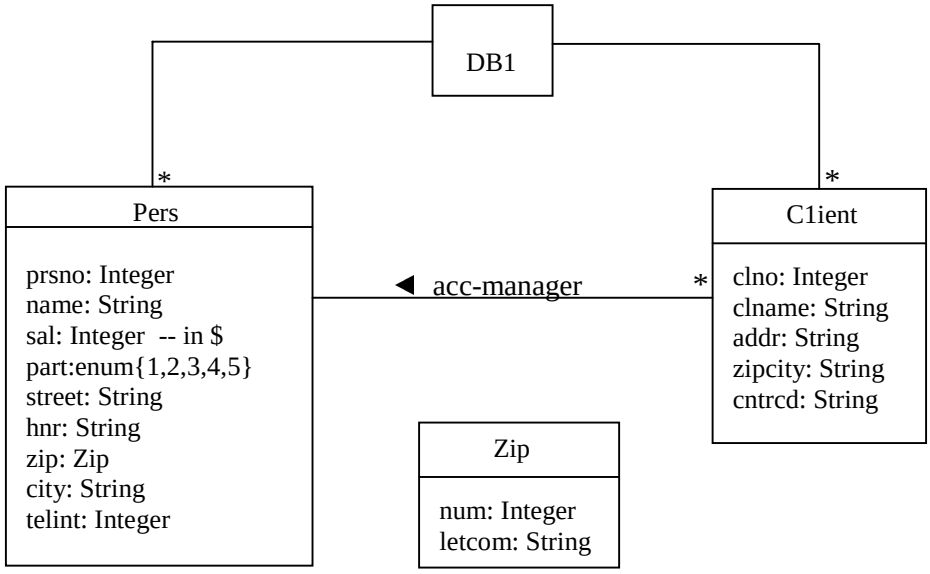
Defining views through constraint definitions is a mistake that is not unusually made in data-modeling practice. This mistake, though understandable, leads to a faulty conception of what a view should constitute. A view should constitute a virtual class, completely derivable in terms of existing base classes in the model, at any given moment and on demand. For this reason, we employ the concept of derived class to represent view definitions in UML/OCL.

4. Component frames

We can also consider a complete collection of databases by looking at so-called component frames, where each (labelled) component is an autonomous database system (typically encountered in legacy environments)



As an example consider a component frame consisting of two separate component database systems: the CRM-database (DB1) and the Sales-database (DB2):



Most of the features of DB1 speak for themselves. We offer a short explanation of some of the less self-explanatory aspects below

- `Pers` is the class of employees responsible for management of client resources
- `part` indicates that employees are allowed to work part time
- `hnr` indicates house number
- `telint` indicates internal telephone number
- `cntrcd` indicates the code of the country the client lives in
- `acc-manager` indicates the employee (account manager) that is responsible for some client's account
- `letcom` indicates a letter combination

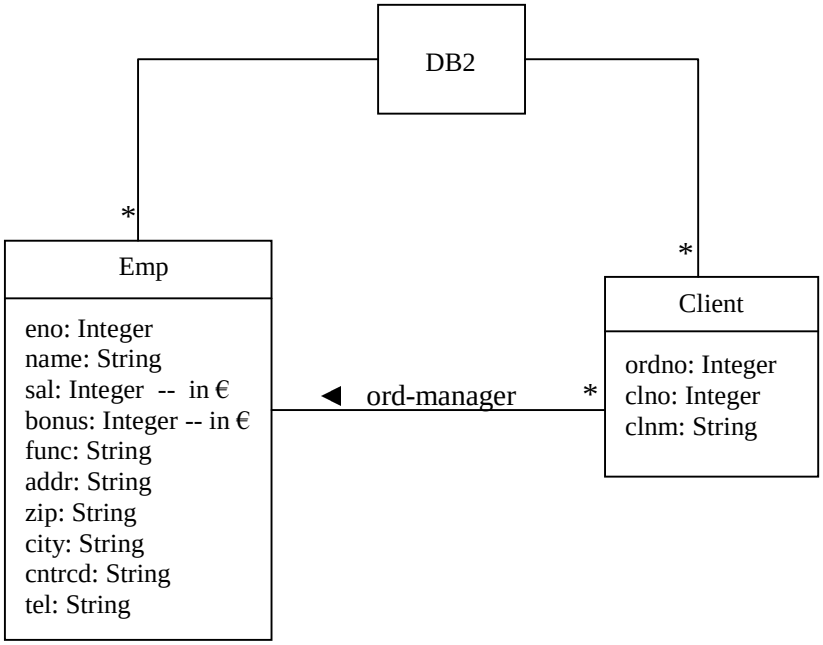
We furthermore assume that database DB1 has the following constraints

```
context Pers inv:
Pers.allInstances --> isUnique (p: Pers | p.prsno)
sal <= 1500
telint >= 1000  and  telint <= 9999

context C1ient inv:
C1ient.allInstances --> isUnique (c: C1ient | c.clno)
cntrcd.size <= 5
```

```
context Zip inv:
num >= 1000  and  num <= 9999
letcom.size = 2
```

The second database is the so-called Sales-database DB2



Most of the features of DB2 also speak for themselves. We offer a short explanation of some of the less self-explanatory aspects below

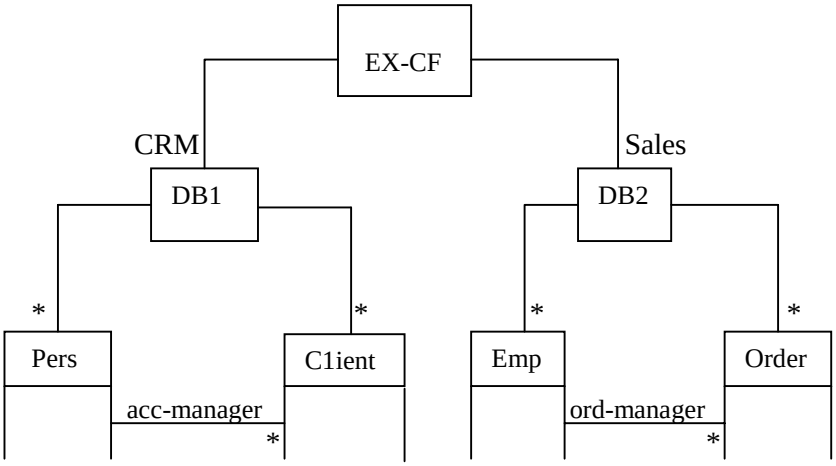
- `Emp` is the class of employees responsible for management of client orders
- `func` indicates that an employee has a certain function within the organization
- `ord-manager` indicates the employee (account manager) that is responsible for some client's order

We assume that this second database has the following constraints:

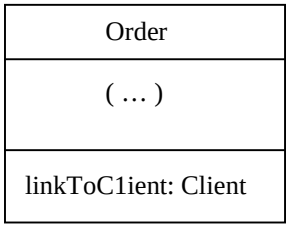
```
context Emp inv:
Emp.allInstances --> isUnique (p: Emp | p.eno)
sal >= 1000
bonus >= 0
tel.size <= 16

context Client inv:
Client.allInstances --> isUnique (c: Client | c.ordno)
Client.allInstances --> forall(c: Client | c.ord-manager.func =
"Sales")
cntrcd.size <= 5
```

The class names `Client` (in DB1) and `Client` (in DB2) happen to be *homonyms*; i.e. the classes have the same names, but also have different meaning. The first Client class refers to a set of clients in a CRM-database. The second class Client refers to a set of client orders, which are maintained in a Sales-database. In order to get rid of confusion, we will perform an first act of *schema cleaning*, by renaming the second Client class to the class Order. We can now place the two databases DB1 and DB2 without confusion into one component frame EX-CF as seen below



The two databases DB1 and DB2 are –in the case of this example- related, in the sense that an order-object residing in class Order in DB2 is associated to a certain client-object in the class C1ient in DB1. On the component frame level, we can define an auxiliary function mapping a order object in class Order to a client object in class C1ient. We do this by assuming an operation in the class Order, called linkToC1ient



with the following post conditions

```
context   Order::linkToC1ient( ): Client
post:     self.linkToClient.clno = self.clno
```

Since the attribute clno has unique values, the link from Order to C1ient is properly defined. (We assume that there always exists a corresponding clno-value in the class Client for each clno-value in the class Order. This is an example of a so-called *inter-database constraint* (also: *component-frame constraint*). We refer to section 11 for more details on this category of constraints.

5. Semantic heterogeneity; the integrated database DBINT

The problems we are facing when trying to integrate the data found in legacy component frames are well-known and are extensively documented (cf. [ShL90]). We will focus on one of the large categories of integration problems coined as *semantic heterogeneity* (cf. [Ver97]). Semantic heterogeneity deals with differences in intended meaning of the various database components. Integration of the source database schemas into one encompassing schema can be a tricky business due to

1. *renaming* (homonyms and synonyms)
2. *data conversion* (different data types for related attributes)
3. *default values* (adding default values for new attributes)
4. *missing attributes* (adding new attributes in order to discriminate between certain class objects)
5. *subclassing* (creation of a common superclass and subsequent accompanying subclasses)

We will offer a general treatment of problems as well as solutions arising in the integration process, by using these above-mentioned five categories of potential conflict situations. We will offer an illustration of problem analysis and accompanying solutions in the context of our example databases.

1. Renaming

By **homonyms** we mean that certain names may –at first sight- look the same (same syntax), but actually have a different meaning (different semantics). **Synonyms**, on the contrary, refer to certain names that are different in the sense that they have a different syntax, but that the actually mean the same (same semantics). Homonyms and synonyms occur extremely often in integration processes. In general, we will adopt the following solution to resolve these naming conflicts: different semantics call for different names, and equal semantics (intended meaning) call for equal names. That is, in the case of two homonyms, we will map the homonyms to two different names. This solution method in the integration process is coined **hom**. An example of two homonyms are the two class names Client (in DB1) and Client (in DB2) in our component frame. We have applied **hom** by creating a class name Order, and subsequently mapping Client (in DB2) to Order, hence distinguishing between class name Client (in DB1) -which remains unchanged- and class name Client (in DB2) -which gets a new name Order.

Synonyms are treated analogously, by mapping two different names to one common name; this solution method in the integration process is coined **syn**. An example of applying **syn** to two synonyms in our database are the attribute names prsno and eno in the classes Pers and Emp, respectively. Integration of these two classes is rather complicated due to the fact that there is only a partial overlap between the two. In a later section we will explain in full how this integration takes place. But in any case, (partial) integration of these two classes into a common class, say PERS, will entail that the attributes prsno and eno are mapped to some common attribute, say pno, having the same semantics, namely that this attribute be a key attribute for the set of class instances of PERS. (In our actual integration of the two classes Pers and Emp, we will construct a common superclass called PERS, and two accompanying

subclasses CRM and SLS, indicating that this superclass PERS reflects the common structure of the related objects residing in the old classes Pers and Emp, while CRM and SLS respectively refer to the discriminating aspects of these related objects. The attribute pno will then be offered a place in this common superclass PERS.)

2. Data conversion

In the integration process, one often encounters the situation where two attributes have the same meaning, but that their domain values are differently represented. For example, the two attributes `sal` in the Pers and the Emp calss of databases DB1 and DB2, respectively, both indicate the salary of an employee, but in the first case the salary is represented in the currency dollars ($), while in the latter case the currency is given in euros (€). What we then do, is to convert the two currencies to a common value (e.g. $, invoking a function `convert€To$`). Another situation is that a *combination* of attributes has the same meaning as some attribute (or combination thereof) somewhere else in the model. For example, the attribute combination of `street` and `hnr` (in Pers) partially has the same meaning as `addr` in Emp (both indicating address values), but the domain values are differently formatted. What we then do is offer some function converting the values of of `street` and `hnr` (in Pers) to a value of `addr` in Emp (cf. section 8, for more details). Applying a conversion function to map to some common value in the integration process, is indicated by **conv**.

3. Default values

Sometimes an attribute in one class is not mentioned in another class, but it could be added there by offering some suitable default value for all objects inside the first class. As an example, consider the attribute `part` in the class Pers (in DB1): it could also be added to the class Emp (in DB2) by stipulating that the default value for all

objects in Emp will be 5 (indicating full-time employment). Applying this principle of adding a default value in the integration process, is indicated by **def**.

4. Missing attributes

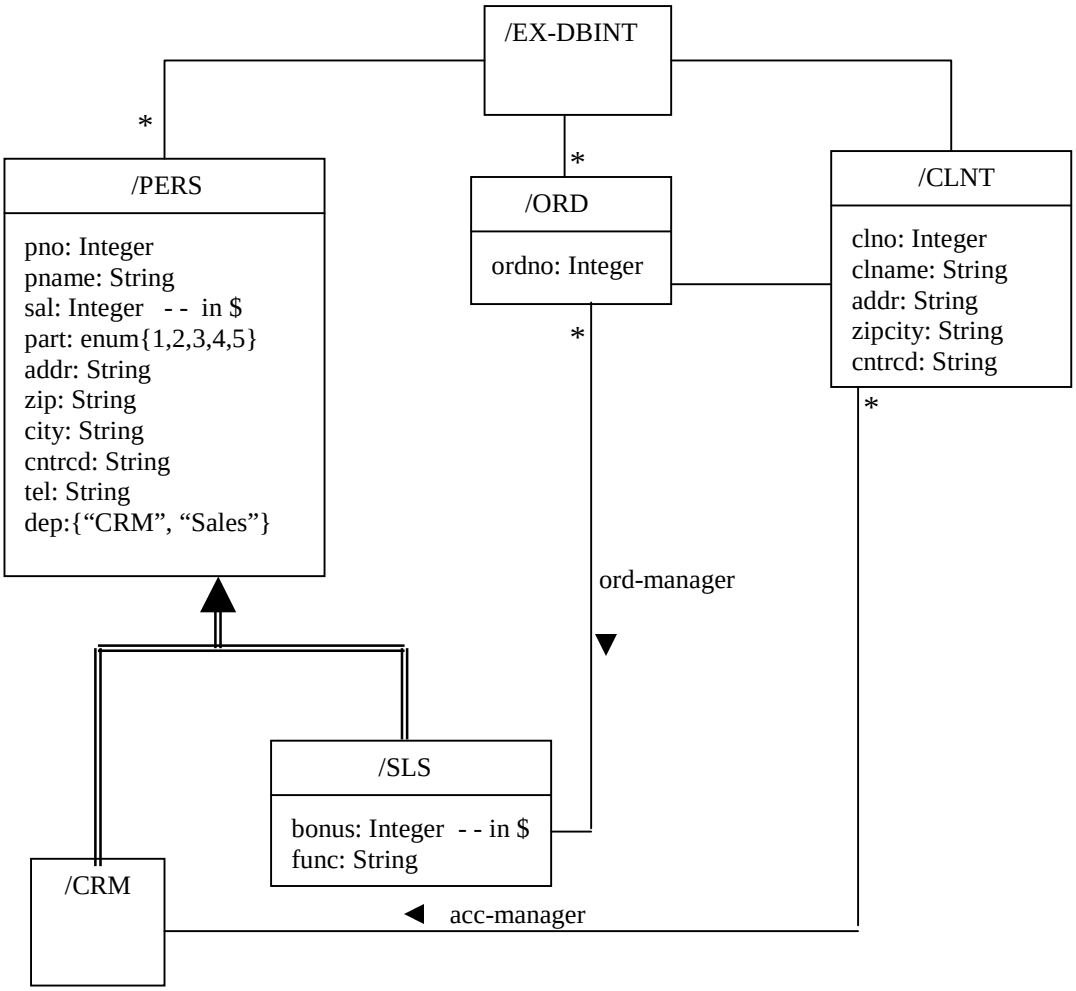
The integration of two classes often calls for the introduction of some additional attribute, necessary for discriminating between objects originally coming from these two classes. This will sometimes be necessary to be able to resolve seemingly conflicting constraints. As an example, consider the classes Pers (in DB1) and Emp (in DB2). Class Pers has as a constraint that salaries are less than 1500 (in $), while class Emp has as a constraint that salaries are at least 1000 (in €). These two constraints seemingly conflict with each other, obstructing integration of the Pers and the Emp class to a common class, say PERS. However, by adding a discriminating attribute `dep` indicating whether the object comes from the CRM or from the SLS department, one can differentiate between two kinds of employees and state the constraint on the integrated level in a suitable manner (cf. section 6 for more details regarding this solution). Applying the principle of adding a discriminating attribute to differentiate between two kinds of objects inside a common class in the integration process, will be indicated by **diff**.

5. Subclassing

The situation of a missing attribute, mostly goes hand in hand with the introduction of appropriate subclasses. For example, introduction of the discriminating attribute `dep` (as described above), entails introduction of two subclasses, say CRM and SLS of the common superclass PERS, by listing the attributes, operations and constraints that are specific to CRM- or SLS-objects inside these two newly introduced subclasses. Applying the principle of adding new subclasses in the integration process, is indicated by **sub**.

6. The integrated database DBINT

We now offer our construction of a virtual database, represented in terms of a derived class in UML/OCL. (For an at length treatment of derived classes in UML/OCL we, again, refer to [Bal02].) The database we describe below, intends to capture the integrated meaning of the features found in the component frame described earlier. We will do so by applying the principles of semantic integration described in the previous section. Consider the following specification of a (virtual) database



This database has the following constraints:

```
context PERS inv:
PERS.allInstances -->
forall(p1, p2: PERS | (p1.dep=p2.dep and p1.pno=p2.pno)
implies
                      p1=p2)
PERS.allInstances -->
forall(p:PERS | p.sal > 1500  implies  p.oclIsTypeOf(SLS))
sal >= 1000.convert€To$
tel.size <= 16
cntrcd.size <= 5

context SLS inv:
bonus >= 0

context CLNT inv:
Clnt.allInstances --> isUnique (c: CLNT | c.clno)
cntrcd.size <= 5

context ORD inv:
Order.allInstances --> isUnique (o: ORD | o.ordno)
```

We shall now carefully analyze the specification of this (integrated) database EX-DBINT, and see if it captures the intended meaning of integrating the classes in the component frame EX-CF and resolves potential integration conflicts.

Analysis:

Conflict 1: Classes Emp and Pers in EX-CF partially overlap, but Emp has no attribute `part` yet, and one still needs to discriminate between the two kinds of class objects (due to specific constraints pertaining to the classes Emp and Pers). Our solution in DBINT is based on applying **syn + def + diff + sub** (map to common class name (`PERS`); add a default value (to the attribute `part`); add an extra discriminating attribute (`dep`); introduce suitable subclasses (`CRM` and `SLS`)).

Conflict 2: Attributes `prsno` and `eno` intend to have the same meaning (a *key constraint*, entailing uniquely identifying values for employees, both for Emp- and Pers- objects). Our solution in DBINT is therefore based on applying **syn** + **diff** (map to common attribute name (`pno`); introduce extra discriminating attribute (`dep`)) and enforce uniqueness of the value combination of the attributes `pno` and `dep`.

Conflict 3: The initial classes Client (in DB1) and Client (in DB2) have different meanings. Our solution is based on applying **hom** (map to different class names). This conflict was already taken care during the stage of determining how to best include both of the Client classes in the component frame EX-CF, where we decided to map the class name Client in DB2 to the class name Order. Hence, this conflict was resolved in a stage prior to the stage of specifying DBINT.

Conflict 4: Attributes `sal` (in Pers) and `sal` (in Emp) partially have the same meaning (salaries), but the currency values are different. Our solution is therefore based on applying **conv** (convert to a common value).

Conflict 5: The attribute combination of `street` and `hnr` (in Pers) partially has the same meaning as `addr` in Emp (both indicating address values), but the domain

values are differently formatted. Our solution is therefore based on applying **syn + conv** (map to common attribute name and convert to common value).

Conflict 6: Attributes `telint` (internal telephone number) and `tel` (general telephone number) partially have the same meaning, but the domain values are differently formatted. Our solution is therefore based on applying **syn + conv** (map to common attribute name and convert to common value).

Resolution of these conflicts is the first step in the actual integration of the classes found in the component frame EX-CF. We are now faced with the subsequent problem to explicitly link the component frame to the integrated (and virtual) database EX-DBINT. We will do so by invoking a so-called *mediator* class.

7. Integrating by mediation

We adopt the so-called *tightly-coupled* approach in integration of a collection of legacy databases into a database federation. This means that we strive at creating a global integrated schema of the federation, which can be queried by a global query language. Tightly-coupled approaches are applicable in relatively stable situations where some form of central data management is involved, such as corporate databases. (For a discussion on so-called *loosely-coupled* versus tightly-coupled systems, we refer the reader to [ShL90].)

Our strategy to integrate a collection of legacy databases –given in some component frame CF- into an integrated database DBINT is based on two principles, being

(1) the tightly-coupled approach to database integration
(2) conformance to the Closed World Assumption of Database Integration (***CWA-INT***)

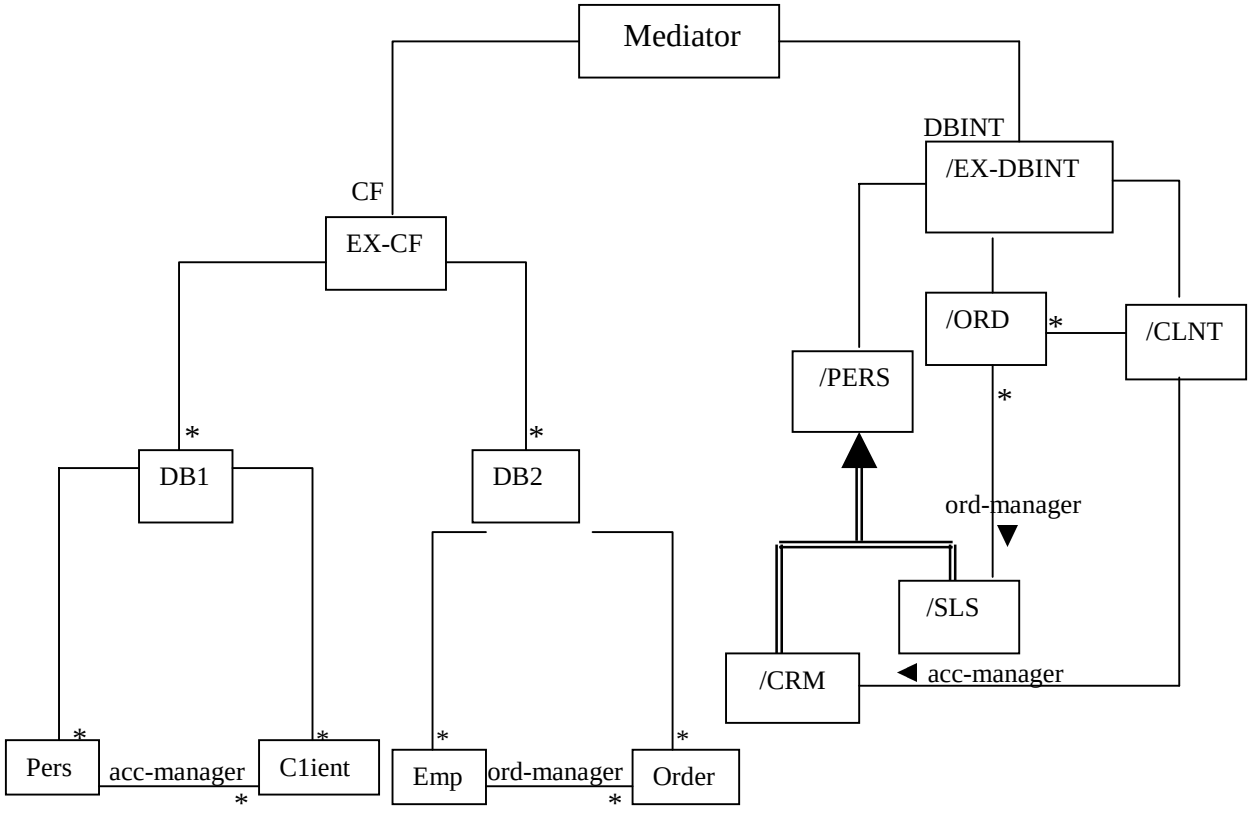
The principle of ***CWA-INT*** can informally be described as follows:

> An integrated database DBINT is intended to hold exactly the "union" of the data in the source databases in CF

Requirement ***CWA-INT*** is a direct extension of the traditional *Closed World Assumption* (CWA) found in the database literature. This assumption (CWA) reads as follows: the only possible instances of a relation are those implied by the database ([Rei84]). In this sense, a database is considered to be complete. Extending CWA to the context of database *integration*, is first discussed in [Hull97], leading to the assumption that we have coined as ***CWA-INT***. This (informal) requirement has to be further investigated for consequences when applied to querying and to updating an integrated database. In more mathematical terms, we will demand that the universe of discourse of component frame CF and the universe of discourse of the integrated database DBINT are, in a mathematical sense, *isomorphic*; only in this way will we not lose any information when transforming the legacy components to the integrated database. (Actually, an *endomorphic embedding* from the universe of discourse of component frame CF and the universe of discourse of the integrated database DBINT will do.) Using conventions taken from OCL, we can describe the universe of discourse of UML model specifications ([WK98]). We will demonstrate, in terms of constraints described in OCL, that the universe of discourse of our example component frame EX-CF and the universe of discourse of the example integrated database EX-DBINT are indeed isomorphic. We refer to section 13 for a description of a general heuristics for realizing such an isomorphism from a component frame to the virtual integrated database. We shall coin this isomorphism as the so-called ***integration isomorphism***.

In this section we will describe a UML model containing a class, called the mediator, explicitly relating the component frame EX-CF and the virtual integrated database EX-DBINT. We will do so, by systematically exploiting various conversion functions, linking objects in the component frame EX-CF to objects in the integrated database EX-DBINT. Constructing these links is done in a very deliberate fashion, with the aim to establish an integration isomorphism between EX-CF and EX-DBINT.

Consider the following model construction, introducing an explicit class Mediator, connecting CF and DBINT



The mediator has the task to correctly link the component frame EX-CF to the (virtual) database EX-DBINT. This is not a trivial task and involves a precise mapping of component elements to the virtual database. The mapping also has to take into account various constraint conditions which rule inside EX-CF. We do this by introducing suitable conversion operations inside the classes.

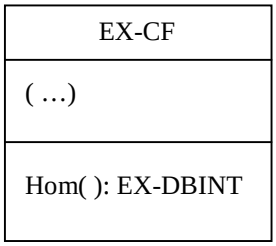
As mentioned earlier, integration of the source database schemas into one encompassing schema can be a tricky business due to the following issues:

1. renaming
2. data conversion
3. default values
4. missing attributes
5. subclassing

We will illustrate that our construction of DBINT (intended to resolve the above-mentioned issues), will actually support ***CWA-INT***. Key to the solution that we offer, is the introduction of a so-called *homogenizing function* which will actually provide for the linking of all relevant features in the component frame to features in the integrated database. This homogenizing function will provide the basis for the integration isomorphism between CF and DBINT that we are looking for.

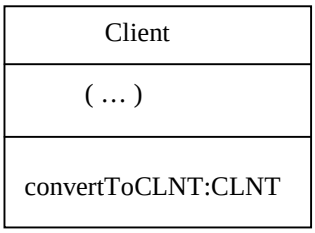
8. Introducing the homogenizing function

In this section we will describe how to add a method, called Hom, to the top-level EX-CF class resulting in an element (database state) of the integrated database EX-DBINT. Hom is the so-called homogenizing function, suitably mapping features of EX-CF to the integrated database EX-DBINT.




```
context    EX-CF::Hom( ):EX-DBINT
post:      self.Hom.CLNT.allInstances =
           (self.CRM.Client.allInstances --> collect(c: Client |
                                                c.convertToClnt))
                                           --> asSet
```

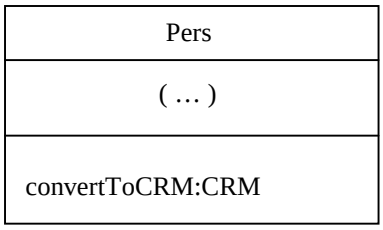
Here we have assumed the existence of a conversion function `convertToClnt` within the class Client:



with the following post conditions

```
context    Client::convertToCLNT( ): CLNT
post:      Client.attributes -->
           forall (d: String | self.convertToCLNT.d = self.d)
           and
          (self.ConvertToCLNT.acc-manager =
           self.acc-manager.convertToCRM)
```

We have now furthermore assumed the existence of a conversion function `convertToCRM` residing within the Pers-class resulting in an object from the class CRM in the DBINT-database

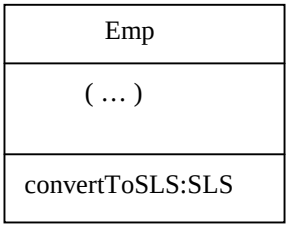


This conversion function has the following post conditions

```
context   Pers::convertToCRM( ): CRM
post:     self.convertToCRM.pno   = self.prsn and
          self.convertToCRM.pname = self.name and
          self.convertToCRM.sal   = self.sal and
          self.convertToCRM.part  = self.part and
          self.convertToCRM.addr  = (self.street).(" ").
                                    concat(self.hnr) and
          self.convertToCRM.zip   = (self.zip.num).(" ").
                                    concat(self.zip.let) and
          self.convertToCRM.city  =  self.city  and
          self.convertToCRM.tel   = ("31-50-363-").(" ").
                                    concat(self.telint) and
          self.convertToCRM.cntrc = "NL" and
          self.convertToCRM.dep   = "CRM"
```

Notice that the function `convertToCRM` is injective!

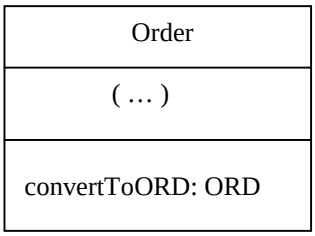
Analogously, we can define a function converting the objects in the Emp-class to corresponding objects in the SLS-class of DBINT, by assuming the existence of a conversion function `convertToSLS` within the class Emp:



with the following (rather trivial) post conditions

```
context   Emp::convertToSLS( ): SLS
post:     self.convertToSLS.pno   = self.eno and
          self.convertToSLS.pname = self.name and
          self.convertToSLS.sal   = self.sal.convert€To$ and
          self.convertToSLS.part  = self.part and
          self.convertToSLS.addr  = self.addr and
          self.convertToSLS.zip   = self.zip and
          self.convertToSLS.city  = self.city and
          self.convertToSLS.tel   = self.tel and
          self.convertToSLS.cntrc = self.cntrc and
          self.convertToSLS.dep   = "SLS" and
          self.convertToSLS.bonus = self.bonus and
          self.convertToSLS.func  = self.func
```

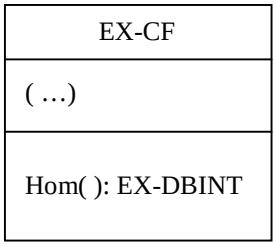
A bit more difficult is the definition of a function converting the objects in the Order-class to corresponding objects in the ORD-class of DBINT. We do this by assuming the existence of a conversion function `convertToORD` within the class Order:



with the following post conditions

```
context   Order::convertToORD( ): ORD
post:     (self.ConvertToORD.ordno =  self.ordno) and
          (self.convertToORD.ord-manager =
          (self.ord-manager).convertToSLS)  and
           self.convertToORD.CLNT =
          (self.linkToC1).convertToClnt
```

where the previously defined operation `linkToC1` provides the link to the unique Client-object associated to a given Order-object.
We now have a complete set of conversion functions mapping objects in the component frame CF to objects in DBINT. The homogenizing function Hom defined in the class EX-CF can now be given its full definition as offered below:



```
context     EX-CF::Hom( ):EX-DBINT
post:       (self.Hom).CLNT.allInstances  =
            (self.CRM.C1ient.allInstances -> collect(c: C1ient |
                                                c.convertToCLNT))
                                             -> asSet) and
            (self.Hom).SLS.allInstances   =
            (self.Sales.Emp.allInstances -> collect(p: Emp |
                                                p.convertToSLS))
                                            -> asSet) and
            (self.Hom).CRM.allInstances   =
            (self.CRM.Pers.allInstances  -> collect(p: Pers |
                                                p.convertToCRM))
                                            -> asSet) and
            (self.Hom).ORD.allInstances   =
            (self.Sales.Order.allInstances -> collect(o: Order |
                                                 o.convertToORD))
                                              -> asSet)
```

With this set of mappings we can define the missing link providing the mapping of objects inside the component frame CF to objects inside the virtual database DBINT. We do this by adding appropriate constraints to the mediator class.

```
context Mediator inv:
self.DBINT.CRM.allInstances   = (self.CF.Hom).CRM.allInstances
and
self.DBINT.SLS.allInstances   = (self.CF.Hom).SLS.allInstances
and
self.DBINT.Clnt.allInstances  = (self.CF.Hom).Clnt.allInstances
and
```

```
self.DBINT.Order.allInstances =
(self.CF.Hom).Order.allInstances
```

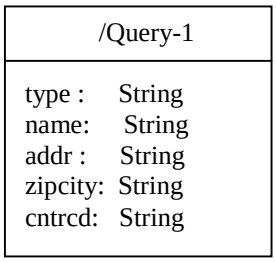
It is now easily verified that the combination of the definition of the homogenizing function together with the constraints offered in the Mediator class, indeed results in an integration isomorphism linking the component frame EX-CF to the integrated database EX-DBINT.

9. Querying the virtual integrated database through the mediator

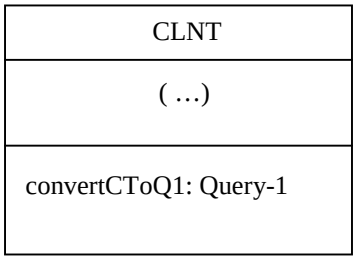
Consider the following example query posed against the integrated database EX-DBINT

"Give the combined list of all clients and CRM-employees"

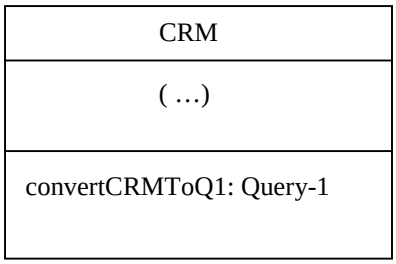
Following [Bal02], a query in UML is specified in terms of a view definition, where a view is conceived as a derived class. We define the following derived class, called `/Query-1`:



Furthermore, we postulate a function convertCToQ1 in the CLNT class, and also a function convertCRMToQ1 in the CRM class



```
context     CLNT::convertCToQ1( ): Query-1
post:       self.convertCToQ1.type    = `CL'  and
            self.convertCToQ1.name    = self.clname  and
            self.convertCToQ1.addr    = self.addr  and
            self.convertCToQ1.zipcity = self.zipcity  and
            self.convertCToQ1.cntrcd  = self.cntrcd
```




```
context     CRM::convertCRMToQ1( ): Query-1
post:       self.convertCRMToQ1.type    = `CRM'  and
            self.convertCRMToQ1.name    = self.pname  and
            self.convertCRMToQ1.addr    = self.addr   and
            self.convertCRMToQ1.zipcity = (self.zip).(" ").
                                          concat(self.city)  and
            self.convertCRMToQ1.cntrcd  = self.cntrcd
```

We then add appropriate constraints to EX-DBINT

```
context  EX-DBINT inv:
Query-1.allInstances =
((CLNT.allInstances --> collect(c : CLNT | c.convertCToQ1))
.Union(CRM.allInstances --> collect(p : CRM |
p.convertCRMToQ1)))
--> asSet
```

By now expanding the definition of CLNT and CRM, we obtain the definition of this query in terms of the original database components found in the component frame EX-CF, but then in terms of the homogenizing function Hom within the context of the *mediator* class (hence , the `self` referred to in the OCL specification below, is the `self` in the context of the class `Mediator`)

```
self.DBINT.Query-1.allInstances =
(((self.CF.Hom).CLNT.allInstances  -->
  collect(c : self.DBINT.CLNT | c.convertCToQ1))
  .Union((self.CF.Hom).CRM.allInstances  -->
  collect(p : self.DBINT.CRM | p.convertCRMToQ1))) --> asSet
```

By expanding the definitions of `(self.CF.Hom).CLNT.allInstances` and `(self.CF.Hom).CRM.allInstances` one level deeper, we obtain the definition of this query in terms of the original components

```
(self.CF.Hom).CLNT.allInstances  =
 (self.CF.CRM.Client.allInstances  ->
  collect(c: self.CF.C1ient| c.convertToClnt)) -> asSet
```

and

```
(self.CF.Hom).CRM.allInstances =
 (self.CRM.Pers.allInstances  ->
  collect(p: self.CF.Pers| p.convertToCRM) -> asSet
```

Hence, the query is now expressed completely in terms of the original database components found in the component frame EX-CF!

The next section concerns actual translation of UML/OCL-specifications of federated database queries to the relational model.

10. Implementing queries on federated databases

In [BP98], a detailed account is given of how to map the basic elements of UML data models to the relational database model (cf. chapters 13 and 14 in [BP98]). Elements such as identity, domains, classes, associations, and inheritance are all systematically mapped to the relational model. The mapping of OCL constraint specifications to the relational model has been investigated in [DH99] and in [DHL01]. In [DH99] the basics are offered of generating SQL-code from database constraints specified in OCL. [DHL01] extends the results offered in [DH99] by investigating how more complex constraints, such as business rules, can be handled explicitly in database applications by means of OCL. Various strategies and experiments with a flexible SQL code generator are discussed, and OCL constraint specifications are evaluated by providing mappings to SQL views.

The results offered in [BP98, DH99] can be extended to databases including views and queries by providing a mapping of derived classes in UML/OCL to the relational database model. This has been demonstrated in [Bal02] by offering a mapping of derived classes in UML/OCL to SQL-code. We remark here that none of these translations from derived class in UML/OCL to SQL-views contain any real surprises, supporting the claim that a general mapping from OCL view constructs to SQL is a more or less straightforward matter.

A complete translation to SQL of our example integrated database EX-DBINT can also be offered along the lines described in [Bal02]. Since this would lead to rather elaborate SQL-code (without, however, containing any real surprises), we refrain from actually doing so here, and we rather refer the interested reader to [Bal02] for more details.

11. Inter-database (component-frame) constraints

Additional information analysis might reveal the following two wishes regarding data in the component frame EX-CF:

(1) Nobody is registered as working for both the CRM and Sales department; i.e., these departments have no employees in common
(2) Client numbers in the Sales database should also be present in the CRM database

This entails that certain constraints should be added, and in this case on the level of the class EX-CF, since these constraints hold between two *databases* DB1 and DB2. Such constraints are called *inter-database constraints* , or *component-frame constraints*. We now offer a specification of the two inter-database constraints mentioned above. We first offer some appropriate abbreviations (using a so-called **let**-construct), and then offer the two constraint specifications.

```
context  EX-CF  inv:
let   P-nrs  = ((self.CRM.Pers.allInstances    ->
                 collect (p:Pers | p.prsno))    -> asSet)
let   E-nrs  = ((self.Sales.Emp.allInstances   ->
                 collect (e:Emp | e.eno))       -> asSet)
let   C-nrs  = ((self.CRM.Client.allInstances  ->
                 collect (c:Client | c.clno))   -> asSet)
let   OC-nrs = ((self.Sales.Order.allInstances ->
                 collect (o:Order | o.clno))    -> asSet)
in
(P-nrs.Intersect(E-nrs)) -> isEmpty  and
 OC-nrs ->
forall(o:Integer| (C-nrs -> exists(c:Integer | o=c)))
```

We are now, of course, also faced with the obligation to suitably introduce this inter-database constraint in the integrated database DBINT. We have already assumed in the construction of EX-DBINT that the second constraint (client numbers in the Sales database should also be present in the CRM database) holds. Hence, we are left with the sole obligation to specify then first constraint, which can be done in a straightforward manner, as illustrated below

```
context  EX-DBINT  inv:
let   X = (self.CRM.allInstances -> collect (c:CRM | c.pno))
                                 -> asSet
let   Y = (self.SLS.allInstances -> collect (s:SLS | s.pno))
                                 -> asSet
in
(X.Intersect(Y)) -> isEmpty
```

As the examples offered above illustrate, OCL offers a powerful means to specify inter-database constraints in a very general manner in the context of our approach.
In the next section, we discuss how to specify so-called *federated updates*; i.e. updates in a federated database system. We shall show that knowledge of component-frame constraints is essential in order to specify federated updates, and that component-frame constraints determine just how loosely- or how tightly-coupled the federation actually is.

12. Updates in a federated database system

Federated updates are defined as updates on a federated database system. These updates can be placed in two large categories: an update involving just one of the component databases in the component frame, or an update involving more than one component database. An example in the first category of updates is the insertion of an order-object in the `Order`-class inside the component `Sales`-database. An example

of the second category of updates is the insertion of a *virtual* `ORD`-object inside the integrated database `DBINT`: such a single insertion on the global level would translate to a whole collection of (local) insertions on the component databases in the component frame CF! In this paper, we will confine ourselves to the first, more simple category of updates, setting out initial guidelines for specifying updates on a federated database. Moreover, confining ourselves to the first category of updates in which we only allow updates on component databases is well within the boundaries of the way that federated databases are often used in practice. Federated systems are in practice often updated solely through the component databases, with the virtual integrated database then used as an on-line integrated global query facility. Furthermore, we shall demonstrate that tackling this problem of strictly *component-confined updates* is interesting enough in itself to deserve separate treatment.

Object insertion

Consider the case that we wish to insert an order-object in the `Order`-class inside the component `Sales`-database of our component frame EX-CF. In order to do so, we will assume that we -somehow- already have an `Order`-object (`o:Order`) at our disposal, which we then wish to insert in the `Order`-class inside the component `Sales`-database of our component frame EX-CF. By this we mean to say that we shall abstract from any `create-` or `new-` like operator for such an `Order`-object, since OCL has no syntax convention for such a create-operator. Hence, we assume that we can invoke an insert operator of the form

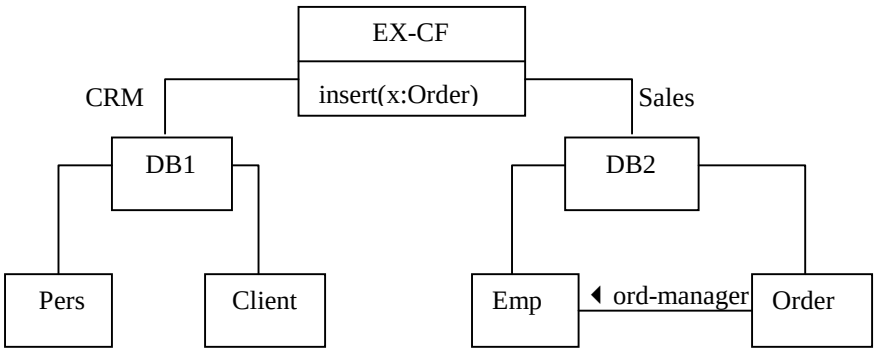
```
insert(x:Order)
```

to actually perform the insertion of a concrete object `o:Order` in the database by applying the insert operation by way of `o.insert`.

The next step is to determine in which class to place such an insert-operator. One might be tempted to place this operator in the `DB2`-class, since it concerns an update of the `Sales` database: success of this update not only has to do with respecting local constraints strictly pertaining to the actual `Order`-class, but possibly also has to do with constraints pertaining to the `Emp`-class (e.g. referential integrity). This consideration, however, has to be taken to an either further consequence, dealing with constraints on the full global level of the component frame! Consider, for example, the second component frame constraint mentioned in the previous section (on inter-database constraints)

(2) *Client numbers in the Sales database should also be present in the CRM database*

This constraint entails that successful insertion of an `Order`-object not only has to respect constraints in `DB2`, but also has to respect a constraint pertaining to the `DB1`-database in the component frame. Hence, the insertion of an `Order`-object actually concerns an update on the level of the component frame EX-CF.
We therefore can conclude that the insert operation is to be placed inside the class EX-CF, as indicated below



We are now left with the task to specify the behavior of this insert-operation. We will do so in terms of the following OCL specification

```
context  EX-CF::insert(x:Order)
pre:     not(self.Sales.Order  -> includes(x))
         and
         (self.CRM.Client -> collect(c:Client| c.clno))->
            -> exists(c:Integer | c = x.clno)
post:    self.Sales.Order = ((self.Sales.Order@pre) ->
                              including(x))
```

The pre-condition of the insert operation consists of two parts; the first part tells us that the `Order`-object has to be new with respect to the set of already occurring instances in the `Order`-class, while the second part says (in accordance with the component-frame constraint (2) mentioned above) that the client number of the `Order`-object should also be present in some object occurrence of the set of instances of the `Client`-class in the CRM database. The post-condition then consists of adding the `Order`-object to the set of instances of the `Order`-class.

We note that when we invoke within the class EX-CF an application of the insert operator `o.insert` on some concrete object `o:Order`, that we assume that this `Order`-object already fully satisfies all constraints on the level of the database `DB2`! This means that we assume that after successful creation of this object all relevant constraint properties within the realm of the component database `DB2` actually hold. Only then will we subsequently consider this `Order`-object as available for other operations (such as our insert operator).

Object deletion

Our next example concerns the deletion of a `Client`-object from the `Client`-class. We will employ the syntax convention `delete(x:Client)` to denote an operation performing this update on the database federation. Again, using similar arguments as in the case of our previous operation `insert(x:Order)`, this delete operation can only be properly placed at the level of the full component frame. This is due to the fact that deleting a `Client`-object could violate the component-frame constraint that an `Order`-object (in database `DB2`) has to refer to an existing `Client`-object (in database `DB1`). The delete-operation can be specified in terms of the following OCL specification

```
context  EX-CF::delete(x:Client)
pre:     not(((self.Sales.Order --> collect(o:Order | o.clno))-
->
         asSet) --> includes(x.clno))
post:    self.CRM.Client = (self.CRM.Client@pre -->
excluding(x))
```

The pre-condition states that an initial check is to be performed ensuring that an `Order`-object does not refer to this particular `Client`-object to be deleted. The post-condition states that `Client`-object is actually removed from the set of instances occurring in the `Client`-class.

Autonomy of component databases

At this stage we wish to say something about so-called *component autonomy* in database federations. In federated database literature it is often claimed that the

component databases should maintain as much as possible their respective autonomy. In practice this makes sense, because a database federation, as we have seen, is actually no more than a database view on a component frame; i.e. the component database remain intact and the federated database is no more than a calculation resulting in a virtual integrated database on the global level. Updating the federated database in actual practice could therefore be considered as updating the component databases. The federated database could be regarded in an even more limited setting by viewing it solely as a means for an integrated query facility on the global level, while updates can only be performed directly on the component databases and *not* via the virtual integrated database on the global level. In this limited setting, one might wish to regard the component databases inside the component frame as autonomous, in the sense that there are no restrictions (with the possible exception of the local database constraints) on allowing for completely autonomous updating of the respective components. This conception of autonomy of the component databases is, however, not without danger. The danger lies in the fact that by allowing a database to become a member of the federation (i.e. the database becomes a component database in a component frame) entails that it might also become subject to certain component-frame constraints! This means –as was the case in our example insert operation- that an autonomous update on a local component database might violate a component-frame constraint, thus eventually rendering it as an incorrect update. Hence, local updates –in a federated setting- are in principle always component-frame updates! It is for this reason that we include an additional check (specified as part of the *pre-condition* in terms of OCL) on the level of component-frame constraints before we engage in actual updating of the database federation.

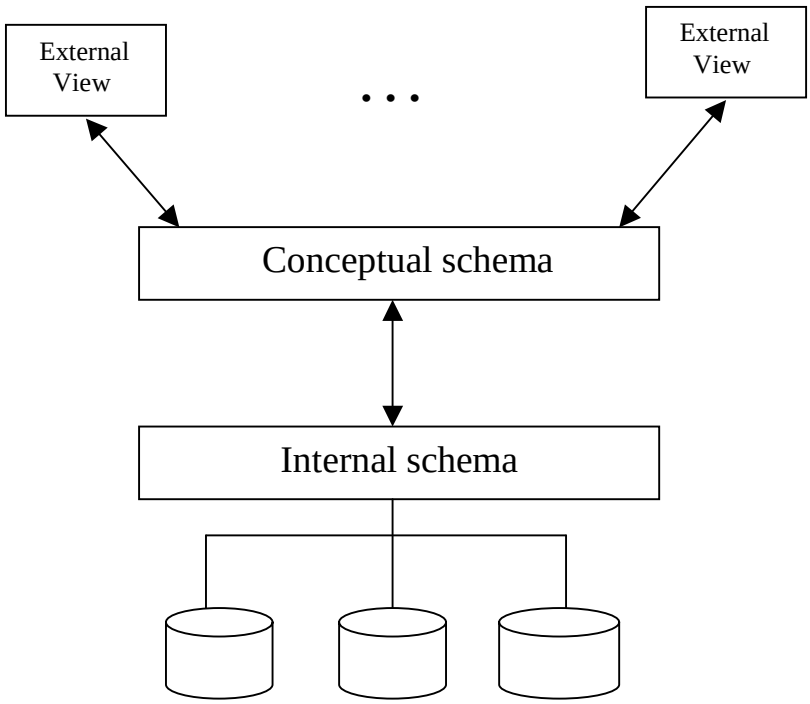
In the next section, we discuss in short an architectural organization of a federated database system based on mediation. In this architectural overview, we will discuss

how to view the logical and physical aspects of a federated database system, with emphasis on how federated architectures relate to the traditional ANSI/SPARC architecture of a monolithic database system.

13. Architecture of a federated database system based on mediation

Traditionally, a monolithic database system is based on what is called the **three-schema architecture** (also known as the ANSI/SPARC architecture), which was proposed to separate user applications and the physical database (cf. [EN00]). In this architecture, schemas can be defined at three levels:

1. The internal level has an **internal schema**, which describes the physical storage structure of the database
2. The conceptual level has a **conceptual schema**, which describes the complete database for the whole community of users. This schema abstracts from physical storage structures, and concentrates on entities, types, relationships, constraints, and operations
3. The external or view level includes a number of **external schemas** or **user views**. Each external schema describes that part of the conceptual schema of the database that is relevant to a particular group of users, and hides other parts that are not relevant to that particular group



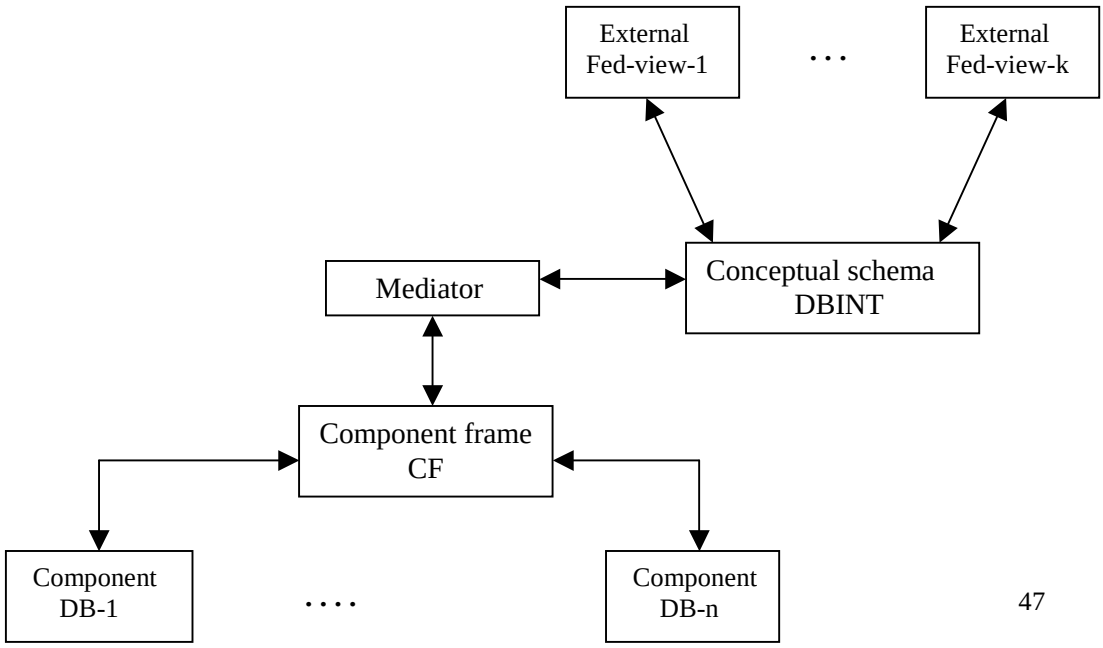
The processes of transforming requests and results between the levels are called **mappings**.

This architecture has the advantage to support the so-called **data-independence property**, meaning that one can change the conceptual schema without having to change the external schema (*logical data independence*), and also that one can change

the internal schema without having to change the conceptual schema (*physical data independence*).

In our setting, we deal with a collection of component databases inside some component frame, with the aim to integrate these component databases, with a federated database as result. As described in section 7, integration is based on the principle of the tightly-coupled approach in combination with the principle of the Closed World Assumption of Database Integration (CWA-INT). In this section we will demonstrate how to achieve an architecture for a federated database, based on these two principles.

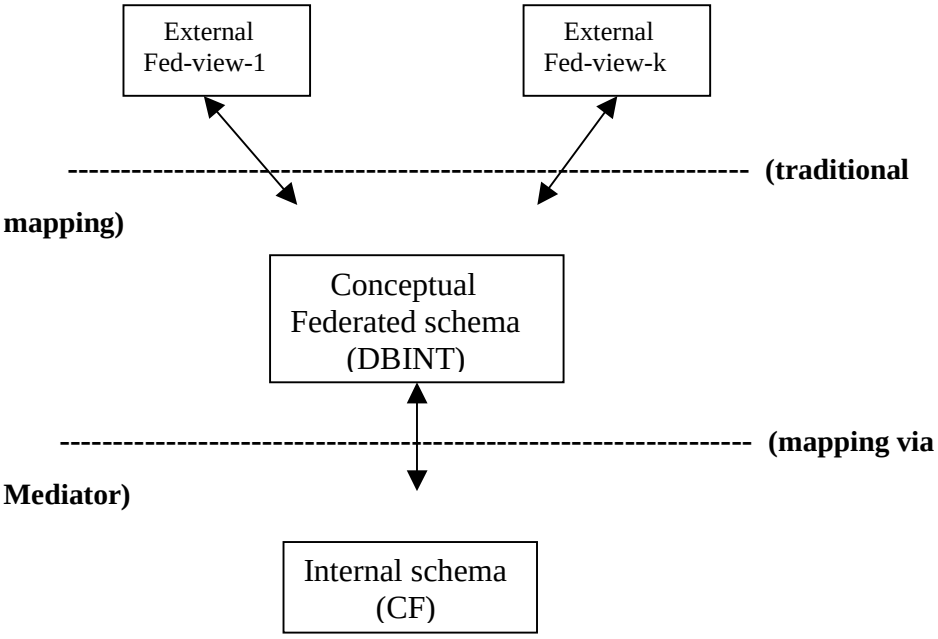
We will assume that each of these component databases internally abide to the three-schema architecture as described above. We are now faced with the problem of what the architecture of the *federated database* looks like. Actually, the solution is quite straightforward. The idea is that the integrated database DBINT contains the conceptual schema of the federation, and that user groups of the federation define user views (with their own separate external schemas) on top of DBINT. We can depict this architecture as follows



where *n* component databases (each abiding internally to their own 3-level architecture) are integrated (via CF and the Mediator), resulting in the database schema of DBINT (representing the conceptual schema of the *database federation*), and where subsequently a number of *k* external views are defined on top of the (conceptual) schema of DBINT. If we succeed in offering a mapping constituting an *integration isomorphism* from the component frame CF to the integrated database DBINT, then we shall also have succeeded in realizing a database federation abiding to the Closed World Assumption CWA-INT; this being our eventual goal of integration.

In this perspective, the architecture of a federated database is basically still much along the lines of a traditional three-level architecture (user views on top of a conceptual schema of a federation, and the eventual internal schema realized via the mediator as a combination of internal schemas of component databases inside a component frame).

We call this architecture a "**three-level federation architecture**", which can be concisely depicted as follows



Analogous to the original three-level architecture, this three-level federation architecture also supports the principles of both logical- and physical data independence. The only difference is that the mapping between the conceptual level and internal level is defined within the context of the database federation, which now is defined via the mediator and the component frame.

14. Heuristics: from specific examples to a general approach

This section concerns a discussion on methodology and the architectural approach, in which we attempt to move from specific examples to a general approach in constructing a database federation from a collection of legacy databases.

As described in the previous section (architecture) and section 7 (mediation as a means to integrate), we adopt the following strategy to integrate a collection of legacy databases (collected in a component frame) into a virtual integrated database

a. create a tightly-coupled architecture of the federated system
b. abide to the principle of the Closed World Assumption of Database Integration (CWA-INT)

Both aspects of this strategy are realized when we adopt the **"three-level federation architecture"** (as described previously) and subsequently establish an **integration isomorphism**, mapping from the component frame to the virtual integrated database. In practice, this can often be a challenging demand, but without succeeding in both aspects, the resulting federated database will fall short due to incorrect query results and inadequate constraint integration.

We now offer some heuristics concerning the realization of the isomorphic mapping from component frame to integrated database. The construction of this isomorphic mapping from the component frame to the virtual integrated database cannot –in principle- be given in algorithmic terms. By this we aim to say that given some set of conflicts in moving from the components to the integrated federated schema, it is usually an illusion to state that there exists an *algorithm* determining how those conflicts are resolved. On the contrary, usually the homogenizing function (***Hom***, in our example) reflects, in terms of a formal specification, the mostly *ad hoc* nature of resolving the conflicts at hand, reflecting the need for a *business semantics* to reach an eventual solution. For example, the resolution of the conflict to establish a common notion for the internal telephone number `telint` (in DB1) and the international telephone number `tel` (in DB2) as given in our example component frame, the homogenizing function Hom introduces the *ad hoc* string

'31-50-363-' (in order to lift the internal phone number to an international phone number). Another example is the conflict of the currencies dollar ($ in DB1) and euro (€ in DB2): deciding which currency is to be taken on the common integrated level is basically *ad hoc,* and has to be offered by the business. This entails that –in general- the process of constructing the formal specification of the homogenizing function Hom (and hence also the isomorphism between the component frame and the virtual integrated database) constantly has to be guided by knowledge of relevant business semantics. Given an arbitrary collection of legacy databases, a general algorithmic solution to arrive at a correctly defined database federation is therefore not feasible. But there is a general heuristics by which this process can be guided.

Equipped with knowledge of relevant business semantics, we can proceed by following a short step-by-step guideline (constituting a heuristics, *not* an algorithm) for constructing a virtual integrated database from a collection of legacy databases, as described below

1. Devise a tightly-coupled architecture for the federation process based on the principles of the **three-level federation architecture**
2. Specify the details of the Component Frame **CF** (possibly with some *schema cleaning*)
3. Analyze semantic heterogeneity: detect conflicts due to *Renaming*, *Data Conversion*, *Default Values*, *Missing Attributes*, and *Subclassing*
4. Construct an integrated schema **DBINT** (applying the principles of **syn**, **hom**, **conv**, **def**, **diff**, and **sub**)
5. Introduce a **mediator class**
6. Enforce ***CWA-INT***, by constructing an ***integration isomorphism*** (via the mediator class) between CF and DBINT based on a suitable ***homogenizing function*** (to be defined in CF)

7. The homogenizing function ***Hom*** is constructed by defining suitable *conversion functions*, mapping component database schemas in CF to new schemas in DBINT
8. Query DBINT by constructing suitable ***derived classes***
9. Add possible **inter-database constraints** in CF, and map to DBINT

Of course, during the process, at some stage it will often be necessary to backtrack to earlier stages to repair choices made in that earlier modeling step. In that sense, this guideline is -in practice- not really step-by-step. Also, as mentioned earlier, this guideline –though systematic- is not algorithmic in nature. Applying the principles set out in this guideline will often demand the necessary creativity from the database modeler, as well as sufficient knowledge of the specific business domain. Apart from these limitations (which apply to most modeling methodologies), our guideline can offer a powerful methodology in moving from a collection of legacy systems to a correctly integrated database system.

Summary

We describe a logical architecture and a general semantic framework for precise specification of so-called database federations. A database federation provides for tight coupling of a collection of heterogeneous component databases into a global integrated system. Our approach to database federation integrates, by means of a so-called homogenizing function, in a uniform and systematic manner the underlying data models of the component systems to a global data model, including constraint specifications. Our focus has been on solving the problems caused by semantic heterogeneity of component systems. The integration process is based on the architectural concept of tight-coupling, and is combined with the so-called Closed World Assumption to establish a notion of union -on the integrated level- of the data

found in the component databases. We have also introduced a special category of constraints, called *inter-database* (or: *component-frame*) constraints, which allow for constraint specifications between the different database components within the federation. The mediating system allows for global queries that can be decomposed in a uniform and systematic manner into local queries on component databases. We also offer a transaction model for a simple set of updates in database federations.

Our approach is based upon the UML/OCL data model. UML is the de facto standard language for analysis and design in object-oriented frameworks, and is being employed more and more for analysis and design of information systems based on databases and their applications. The Object Constraint Language (OCL) - as part of UML - can aid in the unambiguous modelling of database constraints. One of the central notions in database modelling and in constraint specifications is the notion of a database view; a database view closely corresponds to the notion of derived class in UML. We employ OCL and the notion of derived class as a means to treat database constraints and database views in a federated context. The paper demonstrates that our particular mediating system integrates component schemas without loss of constraint information. Furthermore, we offer a setting in which to describe UML/OCL-representations of relational databases.

Acknowledgements:

I wish to thank Bert de Brock of the Faculty of Management and Organization, for numerous discussions, corrections and valuable insights.

References

[AB01] Akehurst, D.H., Bordbar, B.; On Querying UML data models with OCL; «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, 2001, Proceedings. Lecture Notes in Computer Science 2185, Springer, 2001

[Bal02] Balsters, H. ; Derived classes as a basis for views in UML/OCL data models; SOM Research Series 02A47, University of Groningen, 2002

[BB01] Balsters, H., de Brock, E.O.; Towards a general semantic framework for design of federated database systems ; SOM Research Series 01A26, University of Groningen, 2002

[BBZ93] Balsters, H., de By, R.A., Zicari, R.; Sets and constraints in an object-oriented data model; Proceedings Seventh European Conference on Object-Oriented Programming (ECOOP), Kaiserslautern, Germany, July, 1993.

[BP98] Blaha, M., Premerlani, W.; Object-oriented modeling and design for database applications; Prentice Hall, 1998

[BV92] Balsters, H., de Vreeze, C.C.; A semantics of object-oriented sets; Third International Workshop on Database Programming Languages (DBPL; eds. Abiteboul, Kannelakis), Morgan Kaufmann Publishers, California USA, 1992.

[CGW96] S.S. Chawathe, H. Garcia-Molina, J. Widom; A toolkit for constraint maintenance in heterogeneous information systems. 12th International Conference on Data Engineering (ICDE96); IEEE Press, 1996

[Co70] E.F. Codd; A relational model of data for large shared data bank; Communications of the ACM, vol. 13(6), 1970

[DaH84] U. Dayal, H.Y. Hwang; View definition and generalization for database integration in a multidatabase system; IEEE Transactions on Software Engineering 10, 1984

[D00] Date, C.J.; An introduction to database systems; Addison Wesley, 2000

[DH99] Demuth, B., Hussmann, H.; Using UML/OCL constraints for relational

database design; «UML»'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, 1999, Proceedings. Lecture Notes in Computer Science 1723, Springer, 1999

[DHL01] Demuth, B., Hussmann, H., Loecher, S.; OCL as a spevcification language for business rules in database applications; «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, 2001, Proceedings. Lecture Notes in Computer Science 2185, Springer, 2001

[DKM93] P. Drew, R. King, D. McLeod, M. Rusinkievicz, A. Silberschatz; Report of the workshop on semantic heterogeneity and interoperation in multidatabase systems; SIGMOD RECORD 22, 1993

[EN00] R. Elmasri and S.B. Navathe; Fundamentals of database systems; Addison Wesley, 2000

[EP00] Eriksson, H., Penker, M.; Business modeling with UML; OMG 2000

[GR97] Gogolla, M., Richters, M.; On constraints and queries in UML; Proceedings UML'97 Workshop "The Unified Modeling Language – Techniques and Applications", 1997

[GSC96] M. Garcia-Solica, F. Saltor, M.Castellanos; Semantic heterogeneity in multidatabase systems; Object-oriented multidatabase systems; Bukhres, Elmagarid (eds.), Prentice Hall, 1996

[GUW02] Garcia-Molina, H., Ullman, J.D., Widom, J.; Database systems; Prentice Hall, 2002

[Hull97] Hull, R.; Managing Semantic Heterogeneity in Databases; ACM PODS'97, ACM Press 1997.

[Ken91] W. Kent; Solving domain mismatch and schema mismatch problems with an object-oriented database programming language; 7th International Conference on Very Large Databases (VLDB97), 1997

[KoC95] J.L. Ko, A.L.P. Chen; A mapping strategy for querying multiple object databases with a global object schema; IEEE RIDE -DOM,1995

[MC99] Mandel, L., Cengarle, M.V.; On the expressive power of OCL; FM'99 –

Formal Methods, World Congress on Formal Methods in the Development of Computing Science; Lecture Notes in Computer Science 1708, Springer, 1999

[MeY95] W. Meng, C. Yu; Query processing in multidatabase systems; Modern database systems; Kim (ed.), ACM Press, 1995

[OMG99] Object Management Group; Unified Modelling Language Specification, version 1.3; June 1999; **http://omg.org**

[Rei 84] Reiter, R.; Towards a logical reconstruction of relational database theory. In: Brodie, M.L., Mylopoulos, J., Schmidt, J.W.; On conceptual modeling; Springer Verlag, 1984

[ShL90] A.P. Sheth, J.A. Larson; Federated database systems for managing distributed, heterogeneous and autonomous databases; ACM Computing surveys 22, 1990

[SSR94] E. Sciori, M. Siegel, A. Rosenthal; Using semantic values to facilitate interoperability among heterogeneous information systems; ACM Transactions on database systems 19, 1994

[SQL 92] ISO 9075-1992(E); Database language SQL; ISO/IEC JTC1/SC21, 1992

[Ver97] M. Vermeer; Semantic interoperability for legacy databases. Ph.D.-thesis, University of Twente, 1997.

[Wie95] G. Wiederhold; Value-added mediation in large-scale information systems; IFIP Data Semantics (DS-6), 1995

[WK99] Warmer, J.B., Kleppe, A.G.; The object constraint language; Addison Wesley, 1999